H375-EP
S.R

Computers & Security, 18 (1999) 221-230

Certificate Inter-operability — White Paper

John Hughes

Business Development Director – Europe, Entegrity Solutions Ltd, 29 Barnes High St., Barnes, London, SW13 9LW, UK

Purpose of Document

This document is intended to explain the issues and requirements concerning the use of X.509 certificates within 'End Entities', that is users of certificates — such as client workstations or servers. The white paper addresses the following questions:

- What is the purpose of a certificate and what is contained within it?
- Can I use any certificate from any certificate issuer in my SDP application?
- Why do I need Trusted Public Keys?

Certificate Structure

What is a Certificate?

A Certificate is a structure that contains a public value (i.e. a public key) that is bound to an identity. Within a X.509 Certificate the public key is bound to a 'user's name'. A third party (the Certificate Authority) has attested that the public key does belong to the user. When a client receives a certificate from another user the 'strength' of the binding between the public key and identity can vary. This is explained later in the white paper.

A X.509 Certificate is a very formal structure and *can* contain a number of different elements. Those elements that are always contained in a certificate are:

Subject This is the 'user's name' — although it can be any identity value. A number of name spaces are supported. The default is X.500 Distinguished Names (e.g., c=GB, o=Entegrity, cn=hughes). Alternative name spaces supported include RFC822 E-mail addresses (e.g. johnh@entegrity#.com).

Issuer This is the name of the Third Party that issued/generated the certificate, that is the Certificate Authority. The same name spaces are used as defined for the Subject field.

Public Value This is the public key component of a public/private key pair. An associated field defines the public key algorithm being used, for instance whether it is a RSA , Diffie-Hellman or DSA public key.

Validity Two fields are used to define when the certificate is valid from and valid to. Combined together these provide the validity period.

Serial Number This is a field that provides a unique certificate serial number for the issuer.

Signature This is how the Subject identity and the Public Value are bound together. The signature is a digital signature generated by the CA over the whole certificate, using the CA's private key. By having signed the certificate the CA 'certifies' that the Subject is the

'owner' of the public key and has therefore the private key (It is more complicated than this — refer to the Proof of Possession section)

A X.509 Certificate is described using Abstract Syntax Notation 1 (ASN.1) [2,3]. ASN.1 is a both a notation for describing abstracts types and values but also how data is encoded. The encoding scheme is a form of TLV (type length value). That is each field has a Type (for instance to define that the field contains an Integer), the Length of the field, and finally the value of the Field. A field could contain one or more nested fields and hence complicated structures can be constructed. A number of basic types are defined with ASN.1, such a *strings* and *integers*. A number of construction types are also defined with ASN.1, such as *CHOICE* and *SEQUENCE*. The basic and constructor types are usually in-built to an ASN.1 compiler or a ASN.1 library. The in-built types are represented by a tag number, for instance *INTEGER* has a value of 2, *BIT STRING* by 3, *OCTET STRING* by 4, *SEQUENCE* by 16. It is possible to derive user defined ASN.1 constructions out of other user-defined constructions and in-built types. In fact within the context of PKI there is a defined ASN.1 *Certificate* construction. ASN.1 provides the ability for fields to be Optional. For instance one or more fields in a *SEQUENCE* of fields could be defined as being optional. In this case fields are tagged with particular values so that which fields are present are known.

When an ASN.1 encoded certificate (sometimes called a binary certificate) is received then the certificate has to be decoded and its contents extracted. This decoding is performed by an appropriate software module that has been generated either by using an ASN.1 compiler or using a ASN.1 class library (the SDP uses the later technique). During this decoding the software will validate that the certificate is correctly formed.

For example the high level ASN.1 for a Certificate is defined as:

```
Certificate  ::=  SEQUENCE  {
  tbsCertificate       TBSCertificate,
  signatureAlgorithm   AlgorithmIdentifier,
  signature            BIT STRING  }
```

The `TBSCertificate` type is then defined as:

```
TBSCertificate  ::=  SEQUENCE  {
  version                 [0]     EXPLICIT Version DEFAULT v1,
  serialNumber                    CertificateSerialNumber,
  signature                       AlgorithmIdentifier,
  issuer                          Name,
  validity                        Validity,
  subject                         Name,
  subjectPublicKeyInfo            SubjectPublicKeyInfo,
  issuerUniqueID          [1]     IMPLICIT UniqueIdentifier OPTIONAL,
                                     - If present, version must be v2 or v3
  subjectUniqueID         [2]     IMPLICIT UniqueIdentifier OPTIONAL,
                                     - If present, version must be v2 or v3
  extensions              [3]     EXPLICIT Extensions OPTIONAL
                                     - If present, version must be v3
    }
```

Therefore the decoding software will first expect a SEQUENCE field (Certificate is defined as a sequence of various fields). If it receives an unexpected type (that is not the SEQUENCE type) then an error will result. It will then expect another SEQUENCE field (as TBSCertificate is defined as a SEQUENCE of fields). If it does not find a sequence tag an error will ensue. The decoding process will therefore ripple through looking for expected tags, noting of course that some of the tags could be OPTIONAL or a CHOICE.

Whilst processing the ASN.1 structure software will typically extract various fields and place them into memory structures that are easier to process by a programming language. There are a number of ways to achieve this but one could image that ultimately the core fields are placed in some sort of C/C++ structure. Usually the core fields are the basic types defined previously (e.g. INTEGERS, BIT STRINGS). Of course some of the core fields are a collection of basic type fields, and hence the need for C/C++ structures.

Figure 1 illustrates the extraction process. When the decoding software encounters a field that needs to be extracted it will place the value(s) in the C/C++ structure. With the certificate contents having been extracted and placed in the C/C++ structure then a serious of API calls can be provided that allows the calling application to retrieve the appropriate fields. SDP provides a number of methods that permits the caller to encode/decode certificates and to extract/populate them.

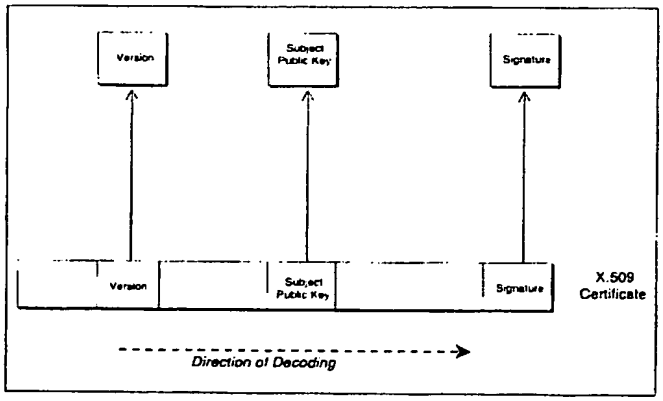

Figure 1 – Decoding extraction process

Another important concept with ASN.1 and PKI in general is that of Object Identifiers, usually referred to as OIDs. OIDs are (usually!) universally unique, their values being allocated by appropriate naming authorities, such as ISO. One can visualize OIDs as being unique type tags, which happen to be more complicated than simple numbers. OIDs are used to uniquely define various fields in ASN.1 structures. For instance the various algorithms used in certificates have defined OIDs. Below is an example of an OID. This is the OID for one of the algorithms used to generate/verify the signature on a Certificate.

```
id-dsa-with-sha1 ID   ::=   {
   iso(1) member-body(2) us(840) x9-57 (10040)
   x9algorithm(4) 3 }
```

When decoding and verifying a certificate the software should recognize the OID and apply the appropriate algorithm. This would allow software to cope with certificates signed using different algorithms.

Certificate Verification

Having decoded a Certificate then the Certificate needs to be verified to ensure that it is valid. This is quite a complicated process and the exact details are outside the scope of this white paper. For details on the certificate path processing refer to [1,4]. The primary processing steps for verifying an End Entities (EE) certificate are:

- Valid (or any) Subject and Issuer names are defined in the EE certificate.
- The EE certificate is not expired by checking the Validity period field.
- The EE certificate has not been revoked (this may be determined by obtaining the current Certificate Revocation List (CRL), current status information, or by out-of-band mechanisms).
- The digital signature on the EE certificate is valid. Remember the digital signature is not verified by using the EE certificate's public key but the issuer's public key. Let us examine this in more detail, but for simplicity let us consider the case of a number of EEs all with certificates issued from a single CA.

Later on we examine cases with multiple CAs. For a general discussion on CA topologies refer to [1]. The steps for validating the digital signature in this case is quite simple, and consists of:

- Extract the issuer's name from the EE certificate.
- Locate the issuer's certificate or the issuer's public key. If you only have the issuer's certificate extract the public key.
- Using the issuer's public key validate that the EE's certificate signature was generated by the issuer

This last step introduces the concept of trust points and trusted public keys. These are discussed later in the white paper.

Are all Certificates the same?

The answer to this is unfortunately ***no!*** Even given that you have a matching EE certificate and related CA (issuer) certificate you may not be able to successfully decode or verify the EE certificate. This can be due to a number of reasons, which are explained below:

Incorrect Implementation: A combination of the technicalities of ASN.1 and X.509 results in a very complicated standard for creating certificates. It is very common for implementers to either incorrectly implement or misinterpret the standard (it is not unknown for standards to be ambiguous!). What is currently missing is a formal means of testing certificates against a reference implementation or having the services of an inter-operability laboratory. What exists is a series of bilateral tests/agreements or groups of vendors working together.

Unsupported Algorithm: You may receive a certificate that has been signed using an algorithm not supported by your software. For instance if you only support RSA , but receive a DSA signed certificate, you will not be able to verify the certificate.

Unsupported Extensions: A Certificate is a very complicated structure and can contain many optional fields. The introduction of version 3 X.509 Certificates added a new sub-structure — that of extensions. A X.509 v3 certificate can have, none, one or more extensions fields. Each extension is defined by an OID. A number of standard extensions are defined by ISO/IETF — but it is also possible for various industry groupings to define their own extensions, whilst still being allocated unique OIDs from an appropriate naming authority. An example of this is the banking community defining their own extensions for defined applications. Extensions can take many forms. One use is to enforce a particular security policy. For instance one extension is called *keyusage*. This extension is used to indicate to the consumer of the certificate the intended use of the public key, for instance digital signature, encryption or key encipherment. X.509 introduces the concept of *criticality*, each extension is marked as to whether it is critical or not, the default being it is not critical. If a certificate is being processed and an extension is detected that is marked critical and the software can't handle the extension then the certificate is to be rejected and the verification fails.

Unsupported Version: There are in fact 3 versions of X.509 certificates, version 1, version 2 and version 3. Version 1 and 3 certificates are more common, with version 3 increasingly so. Whilst Version 3 certificates are a super set of version 1 certificates, it is usually case that version 1 capable software will not be able to process version 3 certificates — however it is normally the case that the opposite situation will work. Using SDP will permit you to process v1 and v3 certificates, however it will only permit you to generate v3 certificates. (see the following section).

X.509 Versions and profiles

X.509, formally known as ITU-T X.509 (formerly CCITT X.509) or ISO/IEC 9594-8, was first published in 1988 as part of the X.500 Directory recommendations. The certificate format in the 1988 standard is called the version 1 (v1) format. When X.500 was revised in 1993, two more fields were added, resulting in the version 2 (v2) format. These two fields are used to support directory access control.

Following experience gained in attempts to deploy PKI systems made it clear that the v1 and v2 certificate formats were deficient in several respects. In response to this, ISO/IEC and ANSI X9 developed the X.509 version 3 (v3) certificate format. The v3 format extends the v2 format by adding provision for extension fields, as previously described. In June 1996, standardization of the basic v3 format was completed [X.509].

ISO/IEC and ANSI X9 have developed standard extensions for use in the v3 extensions field. These extensions can convey such data as additional subject identification information, key attribute information, policy information, and certification path constraints.

Given the complexity and richness of the Certificate together with its various optional fields inter-operability manifests itself as a problem.

> *However, the ISO/IEC and ANSI standard extensions are very broad in their applicability. In order to develop interoperable implementations of X.509 v3 systems for Internet use, it is necessary to specify a profile for use of the X.509 v3 extensions tailored for the Internet. It is one goal of this document to specify a profile for Internet WWW, electronic mail, and IPSEC applications. Environments with additional requirements may build on this profile or may replace it.*[4]

Therefore one aspect of the IETF PKIX working group has been to develop a certificate profile to increase the probability of inter-operability of systems using PKIX conformant certificates. Be aware that most 'legacy' certificate capable software is not PKIX compliant and many current products are still to be upgraded. SDP is one of the few products, at the time of writing at least, that is developed to be PKIX complaint.

Certificate Issuance

Models for Certificate Generation

The previous section explained how having received a certificate one can verify its validity — but how are certificates generated? Whilst certificates are generated at a Certificate Authority (CA) there are two main methods by how this is achieved.

- **Centralized Generation**: The private/public key pair is generated by the CA (or some co-located software) and the public key is directly provided to the CA software to create a certificate. The certificate can then be provided to the consumer (EE or other CA) via any suitable channel. The channel does not have to be secure as a certificate is a self protecting structure (given the CA's signature)
- **Distributed Generation**: In this case the private/public key pair is generated by the EE. The public key is then sent to the CA requesting that it is certified. If the request is valid then the certificate is returned to the requester, and optionally published on some type of certificate repository (e.g. a X.500 Directory).

Of course these two methods can be combined in any system and in reality are, because the trusted CA keys are generated by the CA. For more information on these methods please read [1]. Let is now look at the inter-operability issues for these two methods in the following sections.

Centralized Generation

There are a number of different techniques that can be utilized in this area:

- **Manual Distribution**: In this case the user (i.e. the EE) will be registered on the CA (or associated Registration Authority) by an administrator. Depending on the enterprise's security policy the user may be required to present himself to the administrator. Part of the process of registering the user will be the creation of a token for the user (in PKIX terms this is part of the user's Personal Security Environment — PSE). The token will contain the EE certificate and the associated private key. The token could then be physically supplied to the user. The token could take the form of a disk file or smartcard. For additional security a PIN could be used to 'unlock' the token. If a PIN

system is implemented then it is feasible for the token to be physically posted to the user — or even electronically transmitted. However the PIN should always be sent to the user using a separate secure physical method. Once the EE has received the certificate it may start using its public key to provide security services. This technique does not require the CA to be on-line to the EE's.

- **Request**: Typically in this request the user, using a Web Browser, will connect to a CA's Web Page and request a certificate (or in Verisign's parlance a Digital ID). The user will be prompted to enter some personal details, primarily for identification purposes. The user will also be prompted to enter some form of Pass Phrase. Having requested the certificate (and also triggered the central generation of the public/private key pair) typically the user will be E-mailed with details on how to fetch the certificate. This would be of the form of an E-mail containing a URL of a web page the user must visit to fetch the certificate. On visiting the web site the user would be prompted to enter the Pass Phrase (or something derived from it). The certificate would then be sent to the user using a HTTP message encoded perhaps as a special MIME type that the Web Browser will recognize and be triggered to enter the certificate into the Browser's certificate database. The user would also have to obtain the CA's Trusted Public Key. Most Browsers already come installed with some trusted public keys, for instance Versigns. If the CA's trusted public key is not installed within the Browser then using a similar operation to that described can be used to fetch it (an example of this can be found at www.interclear.net).
- **Request — with authentication**: This is very similar to the previous technique. The additional step is that authentication checks are made. This might take the form of off-line security checks performed on the requester's details. This technique is suited to commercial users of certificates requiring more trust in the certificate.

Distributed Generation

In this method the key material is generated by the EE and the public key is sent to the CA for signing and creation of the certificate. A standalone public key is vulnerable to tampering as it does not have any identity securely associated with it. Therefore the techniques described below are designed to protect the public key in transmission from the EE to the CA.

Before examining the various techniques it is worth outlining the protection mechanisms available. These mechanisms are available no matter the type of transport system used, for example HTTP or E-mail. In summary the methods are:

- **PKCS #10 request: and PKCS #7 response.** Until recently this has been the de-facto standard and is the most widespread. A PKCS#10 certificate request is sent to the CA and a PKCS#7 certificate response is sent back. The PKCS#10 message has a digital signature, which is used to protect the integrity of the request (especially the public key), and provide authentication of the requester. The secure E-mail standard S/MIME is actually based on PKCS#7. SDP supports this protection mechanism.
- **PKCS #10 protected by PKCS #7 request, PKCS #7 response**: This is a variation of the above and is used by Verisign. SDP will soon support this protection mechanism. The PKCS#10 certificate request is also protected within a PKCS#7 message. The PKCS#7 message is encrypted using the CA's trusted public key. Therefore only the CA can decrypt the PKCS#7 message and extract the certificate request.
- **PKIX: using PKIMessage:** This is then new emerging technique for protecting various PKI operational and management messages. SDP supports PKIX protection.
- **Proprietary:** A number of PKI vendors currently implement proprietary mechanisms, for instance Entrust. It is likely that most vendors will adopt PKIX protection in the relatively near future — at least by the end of 1999!

Let is now look at how the above mechanisms can be applied in different situations.

- **Web Browser:** This is basically the same technique as described previously as *'Request'* and

'Request — with authentication'. Without the installation of special Browser applets then the protection mechanisms would normally be PKCS #7/#10.

- **Helper Application:** In this case the PKI vendor would supply a special 'helper application'. Typically this would be a GUI application that allows a user to generate key pairs and request certification of public keys. Entegrity Solutions NotaryClient is an example of this. The transport mechanism would be typically HTTP — in both directions — not relying on any 'out of band' E-mail communication. This technique is more user friendly than the previous Browser described method — but at the cost of having to install the helper application. This technique does depend on having the CA's trusted public key already loaded — perhaps being supplied with the token. This method is described in considerable detail [1].
- **Embedded Application:** The most user friendly method is to hide the PKI from the user completely. To achieve this you need to extend your application to generate key material and request certification. Perhaps this could be performed when an application package is being installed. The installation (or initial start up code) would perform the certification process. Entegrity Solutions Security Development Platform (SDP) is intended to permit application developers to rapidly 'PKI enable' applications to cope with this requirement [5].

Proof of Possession

Before leaving the subject of certificate generation there is one other subject to cover, this is called *'Proof of Possession' (POP)*. In order to prevent certain attacks and to allow a CA/RA to properly check the validity of the binding between an end entity and a key pair, an EE needs to prove that it has possession of (i.e., is able to use) the private key corresponding to the public key for which a certificate is requested. The PKCS#7/#10 protocols do not support POP — however the PKIX protocols does support this as an optional (but highly recommended) feature. The SDP and Notary products support the POP feature when configured to use PKIX.

Trusted Public Keys And Certificates

What are the requirements?

How secure your PKI based application is depends on the 'trust' you can associate with the certificate being used. Fundamentally it depends on whether the public key with the certificate you are using is really 'owned' by the entity (e.g. user) defined by the Subject Name. That is — is this really Alice's certificate? You can have different levels of confidence: in answering this question. There are three primary technical factors that effect the trust.

Below these factors are described and the vulnerabilities that they are protecting against are described below. The different degrees of trust for each factor are also explained.

- **Trust Points:** As described earlier, an EE's certificates are signed by a CA. A certificate verification process involves checking that the CA did actually sign the certificate. This process therefore relies on the EE having obtained the CA certificate. But what protects a CA's certificate — and the EE knowing that it actually is the CA's certificate. Frequently CA certificates are issued in the form of a self-signed certificate. That is the CA's public key is signed using the corresponding CA private key. Whilst this gives integrity protection it does not provide any authentication protection. Any arbitrary entity could produce a key pair and create a self-signed certificate and claim to be, for instance, a Verisign CA certificate. Therefore you need to ensure that you are obtaining a CA certificate from a known source. There are three basic techniques — of various degrees of trust that are explained below:
  - *Known Web Site.* This is the weakest method and involves you downloading the CA trusted certificate from a known web site. Having fetched the certificate then you can load the trusted certificate into the appropriate trusted certificate database. The vulnerability here is that the web site could either be spoofed

(or hacked) and a false CA certificate substituted. You are then liable to attacks where you receive false EE certificates claiming to be from an individual, having been signed by the bogus CA.

— *Embedded certificates.* This technique is prevalent for Web Browsers. Most Web Browsers come pre-loaded with Trusted public keys/certificates, for instance Verisign CA certificates. This is stronger than the 'Known Web Site' technique, but means you are dependent on using a CA prescribed by the Browser product. Whilst this technique is good for 'personal' users it is too inflexible for commercial users or enterprises. For instance it restricts you to support that CA and its supported topology and should the CA's private key be compromised one would have to load a new CA trusted certificate. Some enterprises would wish for the CA to be under their administrative control — which clearly is not the case in this situation.

— *Secure Delivery.* This is the most secure method and provides considerable flexibility. In this case the CA certificate (or just the public key) is provided to the EE by a secure channel. This secure channel could be physical delivery on a Token or via a specially encrypted communications channel (using different key material to the PKI). The physical token technique is the method used by Notary and SDP [1]. This would allow the CA to be under an enterprise's control (or as a Trusted Third Party) and support various CA topologies. PKIX also permits an automated means to automatically update CA trusted public keys should the CA private key become compromised. The Secure Delivery technique is not onerous and is a 'one off' operation per user.

- **Authentication:** When a user submits a certificate request how do you know it actually is the user that it claims to be? I could arbitrarily submit a certificate request supplying a bogus identity and get a certificate issued with the Subject Name being that bogus identity. A number of factors influence this:

  — *No Authentication.* Of course it is perfectly possible to provide no authentication mechanism. This gives no trust.

  — *E-mail Address Authentication.* This is the next weakness approach. Whilst providing the public key to the CA the EE's E-mail address can be provided. The E-mail address can either then be validated (does it exist?) or used to send some type of Pass Phrase to the EE. This scheme relies on not being able to spoof E-mail addresses — which unfortunately is not too difficult to perform.

  — *Pre-registration.* In this case the user is registered on the CA (perhaps by the CA administrator) prior to the certificate request. When the CA receives the certificate request it checks that the EE's identity already exists, if it does then the certificate can be generated. This relies on the fact that attackers can not spoof an identity and also know that the user is pre-registered. It is not a very strong form of authentication.

  — *Pre-registration + PassPhrase.* This is similar to the previous case except the request also supplies a mutually agreed PassPhrase. Depending on the implementation then this method also provides a Proof-of-Possession mechanism. This is a strong authentication mechanism.

  — *Off-line checks.* In this case the CA will perform off-line checks before issuing the certificate, this is not an automated process. It would frequently involve pre-registration and PassPhrases. This is a strong authentication mechanism

- **Proof of Possession:** As described in the previous section a CA should ensure that the requestor does actually possess the private key. One can either not implement this (weak!) or implement the PKIX POP (strong!).

Cross Domain Interactions

Topologies

For simplicity this White Paper has concentrated on describing a CA with a number of subordinate EE's. For a more detailed look at CA hierarchies and cross-certification refer to [1].

However in many Internet situations you may wish to communicate with a EE that has a certificate issued by a CA that is different from your own issuing CA. Below an example is provided that explains what is required.

Example

In the example we have the following:

- **CA A:** This is a CA that has a number of subordinate EE's. (Note that an EE could have a number of parent CAs — it is not restricted to one). CA A has a private and public key pair. The public key, perhaps in the form of a X.509 certificate, is provided to each of its EEs. This is the CA A trusted public key. The private key, held securely on the CA, is used to sign certificates.
- **Alice:** Alice is an EE who has CA A as its CA. Alice possesses the CAs trusted public key, Alice's private key and Alice's certificate (having had its public key signed by CA A to create the certificate).
- **CA B:** This is similar to CA A, except it has its own public/private key pair.
- **Bob:** Bob's parent CA is CA B and has its own key pair and certificate issued by CA B. It will also possess CA B's trusted public key.

Let us now look at the processing steps by which Alice and Bob can communicate with each other securely. To reduce the complexity of the description we will look at the confidentiality (encryption) and authentication (digital signature) services separately. But first let us look at the initialization phase. *Figure 2* illustrates the example and the various steps are 'keyed'. The initialization steps are:

1. Alice generates key material and requests certification of her public key. Her certificate is returned and somehow CA A's trusted public key is provided to Alice. The details of these mechanisms have been previously explained.
2. Bob performs the same process with CA B, obtaining his certificate and CA B's trusted public key.
3. If Alice knows that at sometime in the future she is going to communicate with another user under the control of CA B she will have to load CA B's trusted public key/certificate. This could be an embedded certificate, as previously explained, or loaded by some other means.
4. Bob will perform the same action and ensure he has CA A's trusted public key/certificate loaded.

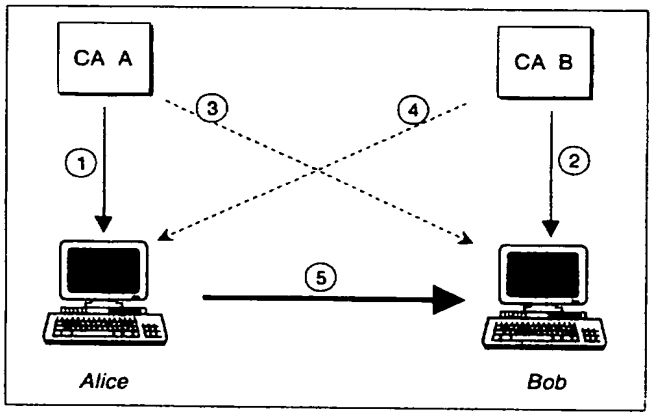

Figure 2 — Cross domain Example

So Alice how has CA A's and CA B's trusted public keys loaded – as does Bob.

First let us take the example of sending a signed E-mail message from Alice to Bob, perhaps using S/MIME. Alice will use her private key to sign the E-mail message. Alice will then send to Bob the signed E-mail message and her EE certificate. On receipt of the e-mail message Bob will perform the following:

1. Verify Alice's EE certificate using the CA A trusted public key. The details of this were explained earlier in the white paper.
2. Having verified Alice's EE certificate the public key is extracted and used to verify the digital signature on the E-mail message.

Encrypted the E-mail message is more complicated and required different processing. The steps to perform this is as follows:

1. Alice will need to fetch Bob's public key. Encryption in this environment involves the originator encrypting the message using the recipient's public key. As only the recipient has the associated private key only he can decrypt the message using his private key. The means by which Alice can obtain Bob's certificate varies. The 'ideal' way of achieving this is by using what is called certificate repositories. When Bob's or Alice's certificates are issued, besides being provided to the requester, they would be published on a certificate repository. Certificate repositories can take the from of LDAP Servers, Web servers or X.500 Directories. Using certificate repositories automates the fetching of certificates. The other method of obtaining Bob's certificate is a manual one — and not exactly user friendly. A typical approach would be for Alice to send E-mail to Bob requesting that Bob send his EE certificate to Alice. In both cases Alice will need to verify Bob's EE certificate using CA B's trusted public key. Alice can then load Bob's EE certificate, extract the public key and use it to protect the E-mail message.
2. Alice will then generate a dynamic symmetric encryption key and use this to encrypt the message. The encryption key is then encrypted using Bob's public key (extracted from his EE certificate).
3. Alice then sends to Bob the encrypted E-mail message and the encrypted symmetric encryption key.
4. On receipt of the message Bob will use his private key to decrypt the encrypted symmetric encryption key. Having recovered the encryption key Bob can then use it to decrypt the encrypted E-mail message.

Conclusions

This white paper has discussed what a certificate is and how it is constructed. It went on to discuss the certificate verification process, explaining what vulnerabilities this process is subject to. The issues surrounding certificate compatibility were discussed and the concept of universally compatible profiles was raised. Certificate issuance and the methods of distribution currently employed were also discussed. Finally, the subject of cross domain interactions, where communication between two parties, who have certificates derived from different certificate issuers was discussed.

It has been seen that each of these processes has certain dependencies, which limit the scope of interoperability — in some cases intentionally. To guarantee inter-operability at a basic low-level the certificates used must have some accepted common profile. This process is currently achieved by interoperability agreements between specific vendors. In some cases, these agreements are limited to targeting cross-certification between two CAs enabling organizations with PKIs supplied by different vendors to establish trusted communications. In other cases, these agreements provide for a tighter coupling of certificate issuance and the ability to recognize a CA of a different vendor as a 'trust point'. The encryption algorithms used must be common to two communicating parties. This point is subject to national policy and export restrictions.

At a higher level, the communicating parties must have a level of trust in the security mechanisms used to issue and maintain their respective certificates, which supports the value of information to be transferred between those organizations.

References

[1] "***Notary and PKI***", Entegrity Solutions White paper. Can be found on www.enetgrity.com web site.

[2] "***ASN.1 — The Tutorial & Reference***", Douglas Steedman, Technology Appraisals Ltd.

[3] "***A Layman's Guide to a Subset of ASN.1, BER and DER***", Burt Kaliski, An RSA Laboratories Technical Note.

[4] "***Internet Public Key Infrastructure — X.509 certificate and CRL Profile***", Section 6. PKIX Working Group Internet Draft.

[5] "***Security Development Platform***", Entegrity Solutions White paper. Can be found on www.enetgrity.com web site.